# What is the Transpose Operation?

*Jon Claerbout*

## ABSTRACT

Geophysical problems generally involve linear operators. Optimization theory demands the transpose to these operators. Examples drawn from NMO, migration, and related areas show that computation of the transpose is usually a straightforward adjunct to the computation itself. An operator may represent only an approximation to reality. Never-the-less, for use in optimization, the computed transpose should be, and generally can be, the exact transpose (within machine precision) of the approximate operator. The transpose is also useful because it is often a practical approximation to the inverse. For example, Kirchhoff migration is really the *transpose* to Kirchhoff modeling, not the *inverse.* Also, the transpose to convolving with a filter response is crosscorrelating with the response. So the transpose isn't the inverse filter, but it does subtract all the phase. The transpose to NMO is much like inverse NMO. The transpose removes all the time shift. NMO itself is trivially invertible when nearest neighbor interpolation is used. One-dimensional stretching deformations, such as NMO, that are done by linear interpolation are easily invertible by means of the tridiagonal solver.

## INTRODUCTION

In geophysics we have a long history of optimization experience with the convolutional filtering operator. Recently, Thorson [1984] introduced optimization to slant stack and velocity stack operators. Nolet [1985] introduced sophisticated inversion theory to the tomographic operator. At SEP we are introducing optimization theory to applications involving NMO, migration, and various kindred operators. A companion paper simultaneously estimates pre- and post-NMO deconvolution filters by applying optimization theory to a physical problem with *three* operators, NMO, filtering, and spherical divergence. Rocca [1982] showed that the midpoint axis can be interpolated by dip moveout operators. Ronen is developing Rocca's theory and may discover that the way to get optimum answers is with optimization theory.

Generally, geophysical theory involves many operators, which may be composited to model a given data set. Inverting the data to a model we need the transpose to the composite operator. This paper illustrates, by means of many examples, how the composite transpose should be computed.

### Matrix multiply

```
if not transpose
    then erase y
if transpose
    then erase x
do iy = 1, ny {
do ix = 1, nx {
    if not transpose
        y(iy) = y(iy) + b(iy,ix) * x(ix)
    if transpose
        x(ix) = x(ix) + b(iy,ix) * y(iy)
    }}
```

### Transient convolution

```
do ix = 1, nx {
do ib = 1, nb {
    iy = ix + ib - 1
    if not transpose
        y(iy) = y(iy) + b(ib) * x(ix)
    if transpose
        x(ix) = x(ix) + b(ib) * y(iy)
    }}
```

## DEFORMATIONS

### Interpolation, nearest neighbor

Let data $x_t = \mathbf{x}$ be resampled about 50% more densely with a crude nearest neighbor scheme denoted by

$$\dot{\mathbf{x}} = \mathbf{B}\,\mathbf{x} \tag{1}$$

where

$$\mathbf{B} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Let us multiply the resampled data $\dot{\mathbf{x}}$ by the transposed resampling operator $\mathbf{B}^T$ and see what we get.

$$\ddot{\mathbf{x}} \;=\; \mathbf{B}^T\,\dot{\mathbf{x}} \;=\; \mathbf{B}^T\,\mathbf{B}\,\mathbf{x} \tag{2}$$

We see that the result $\ddot{\mathbf{x}}$ has the same number of components as the original data $\mathbf{x}$, but it is not exactly equal to the original data. The matrix $\mathbf{B}^T\ \mathbf{B}$ is the diagonal matrix

$$\mathbf{B}^T\ \mathbf{B} \;=\; \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \tag{3}$$

To recover the original data we need to divide by this diagonal matrix.

$$\mathbf{x} \;=\; (\mathbf{B}^T\ \mathbf{B})^{-1}\,\ddot{\mathbf{x}} \tag{4a}$$

$$\mathbf{x} \;=\; (\mathbf{B}^T\ \mathbf{B})^{-1}\,\mathbf{B}^T\ \dot{\mathbf{x}} \tag{4b}$$

$$\mathbf{x} \;=\; (\mathbf{B}^T\ \mathbf{B})^{-1}\,\mathbf{B}^T\ \mathbf{B}\,\mathbf{x} \tag{4c}$$

Curiously, (4b) looks like a familiar equation from least squares theory. Indeed we now understand a theoretical basis for why a transpose operator is often an approximate inverse. It happens in proportion to the diagonal dominance of $\mathbf{B}^T\ \mathbf{B}$.

## NMO nearest neighbor

```
# NMO, transpose NMO, and inverse NMO by nearest neighbor.
# transp=0:    zz(iz) = [NMO]            tt(it)
# transp=1:    tt(it) = [NMO transpose] zz(iz)
# transp=2:    tt(it) = [NMO inverse]   zz(iz)

subroutine nmo1ix( transp, slow, x, t0, dt, nt, tt, zz )
real slow(nt), x, t0, dt, tt(nt), zz(nt), count(4096)
integer  transp, it, nt, iz
real t, x, z, xs, arg
if( transp === 0 )
        do iz = 1,nt
                zz(iz) = 0.0
else
        do it = 1,nt {
                tt  (it) = 0.0
                count(it) = 0.0
                }
```

```
z = t0 + nt * dt
t = z
do iz = nt, 1, -1 {
        xs = x * slow(iz)
        arg = z * z + xs * xs
        # next line replaceable by:  t = sqrt ( arg )
        t = (arg + t * t) / (t + t)
        it = 0.5 + (t - t0) / dt
        if ( it <= nt  ) {
                if( transp == 0 )
                        zz(iz) = zz(iz) + tt(it)
                else {
                        tt(it) = tt(it) + zz(iz)
                        count(it) = count(it) + 1.
                        }
                }
        z = z - dt
        }

if( transp == 2 )
        do it = 1, nt
                if( count(it) != 0.0 )
                        tt(it) = tt(it) / count(it)
return; end
```

## NMO with linear interpolation

The *NMO* transformation is representable as a square matrix, say $\mathbf{B}$. The matrix $\mathbf{B}$ is a $(z, t)$-plane containing all zeros except an interpolation operator centered along the hyperbolic $(z, t)$-trajectory $v^2t^2 = z^2+x^2$. Linear interpolation implies that the matrix $\mathbf{B}$ is a two-band matrix (like a bi-diagonal matrix). Thus $\mathbf{B}^T \mathbf{B}$ is tridiagonal. So if sampling everywhere stretches, the original data can be recovered by solving a tridiagonal system. This idea can be used to program an invertible NMO or trace interpolation.

Using the program below, a field profile was NMOed and then inverse NMOed. The result was then plotted upon the original profile in figure 1. The processed profile is not distinguishable from the original except near the direct arrival. The direct arrival was evidently moved out to before $t = 0$ so it was not recoverable.

Let $\mathbf{T} = NMO^T \; NMO$ denote the symmetric tridiagonal matrix constructed in the program below. Then the pseudoinverse is $\mathbf{T}^{-1} \; NMO^T$. The transpose of the pseudoinverse, namely $NMO \; \mathbf{T}^{-1}$, which is also the pseudoinverse of the transpose, is also an optional output. Since NMO is nearly a unitary operation, you might wonder how *NMO* compares to $NMO \; \mathbf{T}^{-1}$. An example of the two plotted on top of each other is in

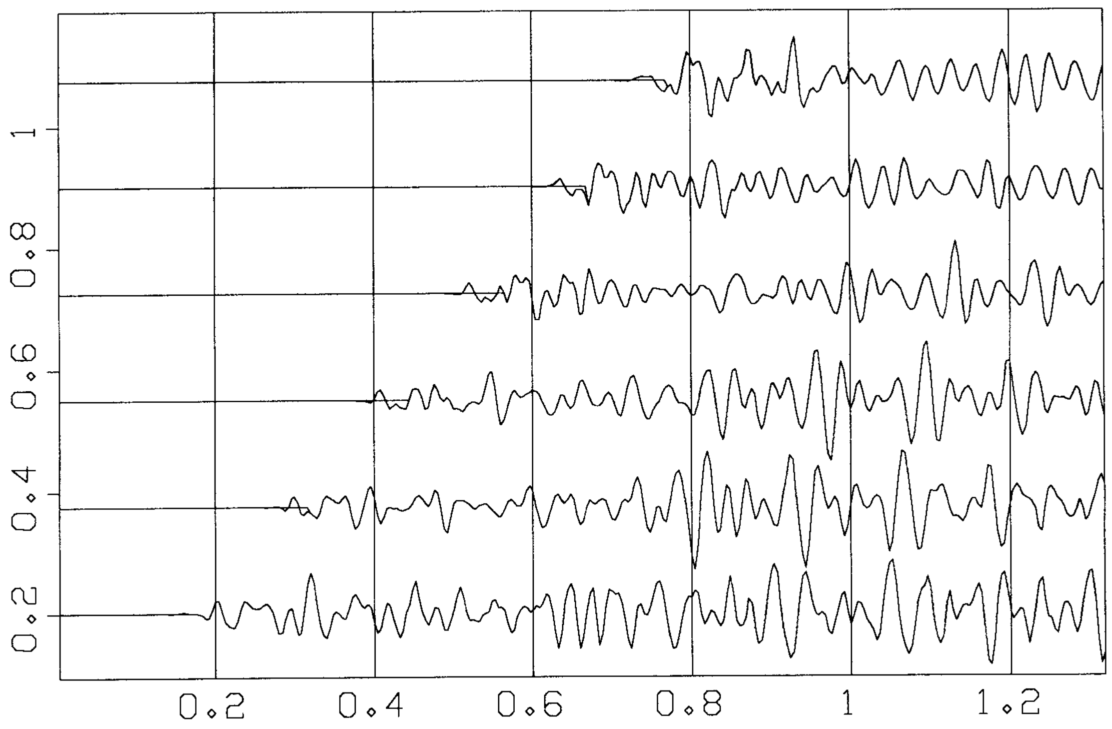

FIG. 1. Field profile $d$ plotted on top of $NMO^{-1}\,NMO\;d$. The two overlay except near the first arrival.

figure 2.

The program below applies an NMO type of operator to a single trace. Notice that tables of interpolation coefficient are constructed and this construction amounts to about 80% of the computational effort. When many traces of the same offset will be (inverse)-(transpose)-NMOed these tables my be reused, thereby saving a factor of five.

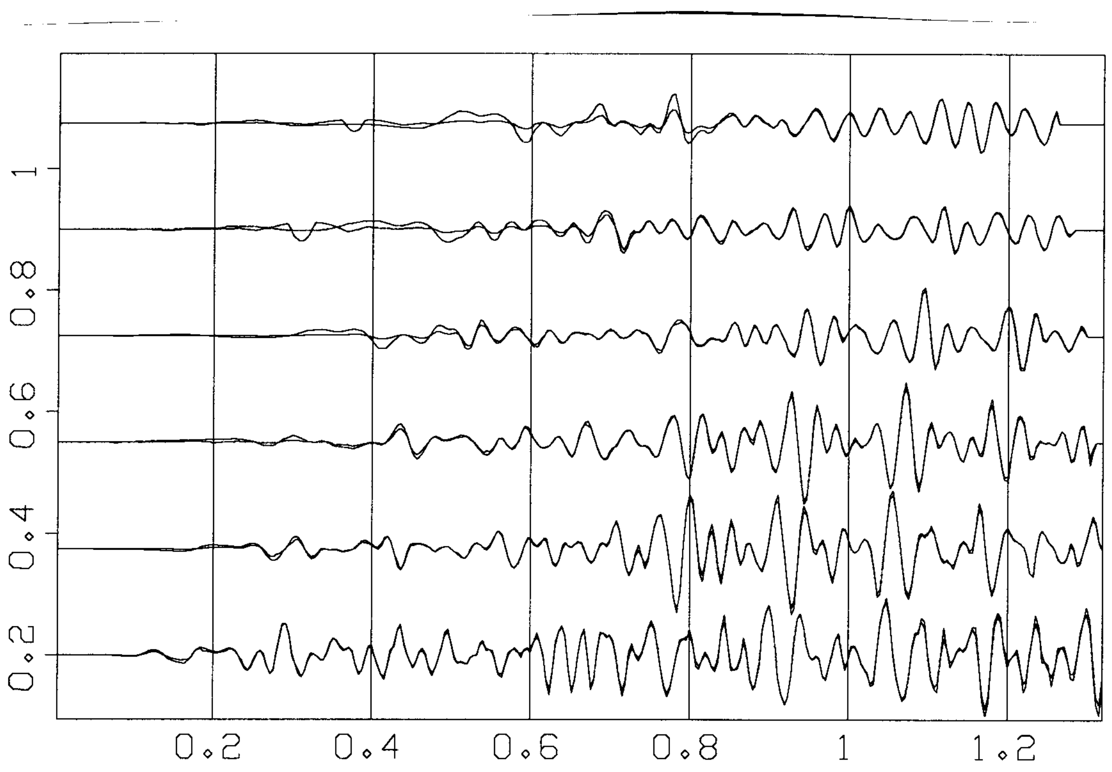

FIG. 2. NMOed field profile *NMO d* plotted on top of $(NMO^T)^{-1}\, d = (NMO^{-1})^T\ d = NMO\ (NMO^T\, NMO)^{-1}\, d$ . The two differ significantly at wide offsets near the first arrival. They also differ noticibly near vertical incidence.

```
# normal moveout a single trace.
# NMO, transpose NMO, and inverse NMO by linear interpolation.
# tra=0 inv=0:zz(iz) = [NMO]                tt(it)
# tra=1 inv=0:tt(it) = [NMO transpose]       zz(iz)
# tra=0 inv=1:tt(it) = [NMO inverse]         zz(iz)
# tra=1 inv=1:zz(iz) = [NMO inverse transpose] tt(it)
```

```
subroutine nmo2ix( tra, inv, mktab, slow, x, t0, dt, nt, tt, zz,
                   itab, bb, cc, w0, w1 )          # work space
real slow(nt), x, t0, dt, tt(nt), zz(nt),
      t, x, z, tm, tpart, xs, arg,
      bb(nt), cc(nt), w0(nt), w1(nt), ts(4096); automatic ts
integer  tra, inv, it, nt, iz, mktab, itab(nt)
if( tra == inv )
      do iz = 1, nt
            zz(iz) = 0.
else
      do it = 1, nt
            tt(it) = 0.
if( inv == 1 )               # initialize tridiagonal coefs.
      do it = 1, nt {
            cc(it) = 0.
            bb(it) = 0.
            }

if( mktab != 0 ) {           # tabulate pointers and weights.
      z = t0 + nt * dt
      t = z
      do iz = nt, 1, -1  {
            xs = x * slow(iz)
            arg = z * z + xs * xs
            # next line replaceable by:  t = sqrt ( arg )
            t = (arg + t * t) / (t + t)
            it = (t - t0) / dt + .00001
            tm = t0 + it * dt
            tpart = t - tm
            w0(iz) = (dt - tpart) / dt
            w1(iz) = 1. - w0(iz)
            itab(iz) = 0
            if ( it+1 <= nt ) {          # interior
                  itab(iz)  = it
                  bb(it)    = bb(it)   + w0(iz) * w0(iz)
                  cc(it)    = cc(it)   + w1(iz) * w0(iz)
                  bb(it+1) = bb(it+1) + w1(iz) * w1(iz)
                  }
            else if ( it <= nt ) {       # at edge
                  itab(iz)  =  - it
                  bb(it)    = bb(it)   + w0(iz) * w0(iz)
                  cc(it)    = cc(it)   + w1(iz) * w0(iz)
                  }
            else                         # off end
                  itab(iz) = 0
            z = z - dt
            }
      }
```

```
do it = 1, nt                      # avoid destruction of input.
        ts(it) = tt(it)
if( tra == 1  & inv == 1 ) {     # transpose pseudoinverse
        do it = 1, nt
                if( bb(it) == 0.0 ) {
                        bb(it) = 1.0
                        ts(it) = 0.0
                        }
        call vtris(nt, cc, bb, cc, ts, ts)   # vtris allows overlay.
        }

if( tra == inv ) {        # Operator itself or transpose pseudoinverse
        do iz = 1, nt {
                it = itab(iz)
                if( it > 0 ) {
                        zz(iz) = zz(iz) + w0(iz) * ts(it)
                        zz(iz) = zz(iz) + w1(iz) * ts(it+1)
                        }
                else if( it < 0 ) {
                        it = -it
                        zz(iz) = zz(iz) + w0(iz) * ts(it)
                        }
                }
        }

else {                  # transpose or inverse
        do iz = 1, nt {
                it = itab(iz)
                if( it > 0 ) {
                        ts(it)   = ts(it)   + w0(iz) * zz(iz)
                        ts(it+1) = ts(it+1) + w1(iz) * zz(iz)
                        }
                else if( it < 0 ) {
                        it = -it
                        ts(it)   = ts(it)   + w0(iz) * zz(iz)
                        }
                }
        }

if( tra == 0  &  inv == 1 ) {   # pseudoinverse
        do it = 1, nt
                if( bb(it) == 0.0 ) {
                        bb(it) = 1.0
                        ts(it) = 0.0
                        }
        call vtris(nt, cc, bb, cc, ts, tt)
        }
else
        do it = 1, nt
                tt(it) = ts(it)
return;         end
```

You may wonder why the program bothers to distinguish time truncation at the first linear interpolation coefficient from truncation at the second. I found the simpler truncation logic often gave zero division in the tridiagonal solver, thereby preventing the inversion. The matrix $A$ is almost square, depending how we do the truncation. It seems that if we plan to invert $A^T A$, we should truncate the last column of $A$ rather than the last row.

```
# variable coefficient tridiagonal solver adapted from FGDP.
# FGDP:        C(K  ) * T(K-1) + B(K) * T(K) + A(K) * T(K+1) = D(K)
# here:        c(K-1) * T(K-1) + B(K) * T(K) + A(K) * T(K+1) = D(K)
# So for a symmetric matrix:  c(K) = A(K)

subroutine vtris(n, a, b, c, d, t)
integer n, k
real a(n), b(n), c(n), t(n), d(n), e(1024), f(1024)
e(1) = - a(1) / b(1)
do k = 2, n-1
        e(k) = - a(k) / ( b(k) + c(k-1) * e(k-1) )
f(1) =  d(1) / b(1)
do k = 2, n
        f(k) = ( d(k) - c(k-1) * f(k-1) ) / ( b(k) + c(k-1) * e(k-1) )
t(n) = f(n)
do k = n-1, 1, -1
        t(k) = e(k) * t(k+1) + f(k)
return;          end
```

## Resampling in offset space

If a CDP gather is to be resampled in offset, the interpolation should be along hyperbolic trajectories. (The hyperbola varies with $\tau$ according to $v(\tau)$). I use nearest neighbor on the $t$-axis and linear interpolation on the $x$-axis. Interpolation along hyperbolic trajectories is a multidimensional transformation. Since we don't have a multidimensional tridiagonal simultaneous equation solver, we don't have an exact inverse to the interpolation along hyperbolic trajectories. If you want an invertible transformation, you should first use "nmo", then laterally stretch with "midpoint".

## Radial trace

The basic equations for radial trace transformation are:

$$t^2 = \tau^2 + x^2 / v(\tau)^2 \qquad (5)$$

$$u = x / \tau \qquad (6)$$

$$t^2 = \tau^2 (1 + u^2 / v(\tau)^2 ) \qquad (7)$$

Below is the central part of my program. The inverse is not everywhere exact when $v = v(\tau)$.

```
do iu=1,nu {
        u = u0 + iu*du
        tanang = u * slow(iu)
        cosi =  sqrt( 1. + tanang * tanang )
        do iz=1,nz {
                z = z0 + iz*dz
                x = u * z
                ix = (x - x0) / dx + .5
                t = z * cosi
                it = (t - t0) / dt + .5
                if( 0 < ix  &  ix <= nx  &  0 < it  &  it <= nt ) {
                        if( transp == 0 )
                                uu(iz,iu) = uu(iz,iu) + xx(it,ix)
                        else {
                                xx(it,ix) = xx(it,ix) + uu(iz,iu)
                                count(it,ix) = count(it,ix) + 1.
                                }
                        }
                }
        }
```

Since the program is only invertible where stretching, I usually define $d\tau$ (=dz) somewhat smaller than $dt$ .

## Typical case of deformation transformation

Below is a sketch of a generalized deformation transformation, its transpose, and a pseudoinverse based on getting proper scaling of the diagonal of $A^T A$ .

```
if  not transpose
        then erase y
if  transpose
        then erase x

Loops over output space i-indices {
        input j-indices as functions of i-indices
        Summation or interpolation loops {
        if j-index on data {
                value = any function of indices
                if  not transpose
                        y(i1,i2...) = y(i1,i2...) + value * x(j1,j2...)
                if  transpose
                        x(j1,j2...) = x(j1,j2...) + value * y(i1,i2...)
                        sum(j1,j2...) = sum(j1,j2) + value
                } } }
```

```
if pseudo inverse
    loop over j-space
        if sum(j1,j2,...) $\neq$ 0.0
            x(j1,j2,...) = x(j1,j2,...) / sum(j1,j2,...)
```

At a later time I will want to figure out how to get the best unitary approximations by row and column scaling. We should study to see how to make a tridiagonal approximation for any one-dimensional pseudoinverse.

## Inner product tests

Paige and Saunders in their paper on optimization point out that their program fails to converge when the given operator and its given transpose are not really transposes of each other. I run so few iterations this test was never invoked. I asked Stew Levin for a test that an operator really is the transpose of another operator. He suggested $y(Ax) = (yA)x$. For example, $x$ could be data from the South Atlantic and $y$ could be data from the North Pacific. Then $Ax$ could be moveout corrected South Atlantic data, and $yA$ would be inverse moveout corrected North Pacific data. The following lines were output from my test program:

```
"nmo1"
    dotprod=-55845879201923072.
    dotprod=-55845866317021184.
"nmo2"
    dotprod=-50359445028339712.
    dotprod=-50359483683045376.
"radial"
    dotprod=63642576284024832.
    dotprod=63642309996052480.
"midpoint"
    dotprod=-48595940046536704.
    dotprod=-48595622218956800.
"offset"
    dotprod=-48774632160886784.
    dotprod=-48774262793699328.
```

So we see the departures occur in the sixth significant place.

I found these dot product tests shook a lot of bugs out of my header processing procedures. Be aware that the NMO process depends on the geometry of the data layout, so the layout better be the same in the North Pacific as it is in the South Atlantic. It is noteworthy that a matrix need not be square. The last three processes tested above produced more output than they had input. The test also shook some bugs out of my scientific programs so I plan to continue to make the test on each process I program.

### Stolt migration and diffraction

NMO is based on the quadratic equation $v^2 t^2 = z^2 + x^2$. Stolt's migration, he noted in SEP 20, is also based on a quadratic equation $\omega^2/v^2 = {k_z}^2 + {k_x}^2$. So Stolt migration is NMO in the Fourier domain.

$$\mathit{Stolt} \quad = \quad FT \ \rightarrow \ \mathit{NMO} \ \rightarrow \ FT^{-1} \tag{8}$$

A property of matrix transposes is $(\mathbf{A}\,\mathbf{B}\,\mathbf{C})^T = \mathbf{C}^T\,\mathbf{B}^T\,\mathbf{A}^T$. We know the transpose of NMO, and we know that the (conjugate) transpose of Fourier transform is inverse Fourier transform. So

$$\mathit{Stolt}^{\,T} \quad = \quad FT \ \rightarrow \ \mathit{NMO}^{\,T} \ \rightarrow \ FT^{-1} \tag{9}$$

So we see the transpose to Stolt modeling is Stolt migration, provided that you forget the Jacobian, as I usually do.

Of course the NMO program must work with complex values. I wonder if there are any time-domain applications for the complex-valued NMO program. Some of the end effects and interpolation problems are challenging. Bill Harlan showed that doing better than linear interpolation saves need of a lot of zero padding. On the other hand, linear interpolation is nicely invertible by the tridiagonal system. I guess we'll have to dust off the general band matrix solver.

## MIGRATION AND DIFFRACTION

The linear interpolation within a deformation transformation is a weighted sum over two points. In Kirchhoff migration and diffraction, the summation is over more than two points.

The transpose to summation is taking a point and spreading it around, actually, adding it atop of what is already out there. I'll refer to the transpose of summation as "spraying".

### Kirchhoff modeling and migration.

Ignoring velocity and "if index off data" tests, Kirchhoff modeling and migration is:

```
do iz = 1,nz
    do ih = —25, 25
        it = sqrt( iz*iz + ih*ih )
        do iy = 1,ny
            ig = iy + ih
            if not transpose
                zz(iz,iy) = zz(iz,iy) + tt(it,ig)     # summing
            if transpose
                tt(it,ig) = tt(it,ig) + zz(iz,iy)     # spraying
```

The trick to making the Kirchhoff fast is to move the y-loop to the inside of the square root and interpolation overheads.

**Gazdag migration and diffraction**

The Gazdag algorithm is the most complicated one considered so far. Below you need to check two things, is it really the Gazdag modeling and migration program? Are the operations really the transposes?

A property of matrix transposes is $(\mathbf{A}\,\mathbf{B}\,\mathbf{C})^T = \mathbf{C}^T\,\mathbf{B}^T\,\mathbf{A}^T$. Interpreted in terms of layers, this means if the operator goes from the earth's surface to its interior, then the transpose operator goes from the interior to the surface. Observe in the program sketch below that the loops run in opposite directions, the inverse Fourier transform is its transpose, and the order of *FT* , *layer* , and *source* are reversed in the transpose.

```
if not transpose          i.e. migration
    erase all of $Image(z, k_x)$
    $U(\omega, k_x) = FT2D[u(t, x)]$
    for $z = 0,\ z < z_{\max},\ z = z + \Delta z$
        call $source$
        call $layer$
        $image(z, x) = FT^{-1}[Image(z, k_x)]$
if transpose                    i.e. diffraction
    erase all of $U(\omega, k_x)$
    for $z = z_{\max},\ Z > 0,\ z = z - \Delta z$
        $Image(z, k_x) = FT[image(z, x)]$
        call $layer$
        call $source$
    $u(t, x) = IFT2D[U(\omega, k_x)]$
return; end
```

The *layer* subroutine resembles a diagonal matrix multiply.

```
subroutine layer
for all $\omega$ and all $k_x$
    $C = \exp( - \Delta z \sqrt{(-i\,\omega)^2/v^2 + k_x^2} )$
    if not transpose
        $U(\omega, k_x) = U(\omega, k_x) * \overline{C}$
    if transpose
        $U(\omega, k_x) = U(\omega, k_x) * C$
return; end
```

The *source* subroutine next displays a typical transpose character. In modeling, the image, being a point in time, is sprayed out into all frequencies. In migration, the image point is created by summing all frequencies.

```
subroutine source
for all $\omega$ and all $k_x$
    if not transpose
        $Image(z, k_x) = Image(z, k_x) + U(\omega, k_x)$
    if transpose
        $U(\omega, k_x) = U(\omega, k_x) + Image(z, k_x)$
return; end
```

The above program sketch looks correct, but no real program for it has yet gone through the $(yA)x = y(Ax)$ test.

## Finite difference wave extrapolation, $(\omega, x)$

Let us see what needs to be done to the program in IEI p 105 to incorporate the transpose operator. First, the velocity is a constant function of $z$ so it isn't absolutely necessary to reverse the $z$-loop. Second, the program contains the three lines

```
do ix=2,nx-1
        cd(ix) = aa*q(ix+1) + (1-2*aa)*q(ix) + aa*q(ix)
call ctris
```

In the transpose operation, you expect to swap the call and the do loop. But here, I believe everything is a power of a tridiagonal matrix, so it all commutes.

Another area of discrepancy between the forward and the inverse program arises with the side boundaries. I believe Dave Hale's coding of the side boundaries, on page 106 translates nicely to its conjugate, so it should work in either the forward or the transpose program.

It seems the principle change to get the conjugate transpose program is to replace "aa" by its complex conjugate, which makes the waves go the other way.

### Integration

The causal integration operator is like a matrix with ones below the diagonal and zeros above. So the transpose to causal integration is anti-causal integration. The $Z$-transform expression for causal integration is $(1 + \rho Z)/(1 - \rho Z)$. The transpose operator is $(1 + \rho/Z)/(1 - \rho/Z)$

### Recursive dip filters

Recursive dip filters are phaseless in $x$, so I guess the 2-axis spatial filters are their own transpose, though I will want to check the side boundaries in my program. Transposed dip filters must do their time-domain recursion anticausally.

### More exercises for the reader

1. $15^\circ$ time-domain migration
2. $45^\circ$ migration
3. Bullet-proof migration
4. Migration with absorbing sides
5. dip moveout.

### Programming style

As a matter of programming style, an objection has been raised to putting an "if transpose" test in the inner loop of a program because it increases computational cost somewhat. On the other hand, the advantage of having one program, compared to having two is that improvements to the model are immediately seen in the transpose and vice versa. There is no need to update two separate programs and assure their continued consistency. A compromise is to move the "if transpose" back up one loop.

## CONCLUSION

It seems to be a straightforward matter to compute the transpose operation within the same program that applies the operator itself. By doing both operator and transpose in the same program, the same approximations and truncations are made and the chances are excellent that the transpose will be exact within machine precision.

Theoretical studies that determine an analytic representation for the transpose are not required, but such theoretical results might help us define operators whose transpose more nearly approximates their pseudoinverse.

## REFERENCES

Bolondi, G., Loinger, E., and Rocca, F., 1982, Offset continuation of seismic sections: Geophys. Prosp. **30,** 813-828.

Claerbout, J.F., 1985, Imaging the earth's interior: Blackwell Scientific Publications.

Luenberger, David G, 1973, Introduction to linear and nonlinear programming, Addison-Wesley

Nolet, G., 1983, Resolution analysis in large scale tomographic systems: lecture given at Stanford University

Nolet, G., 1985, Solving or resolving inadequate and noisy tomographic systems: submitted to J. Comp. Phys.

Paige, C.C. and Michael A. Saunders, 1982 LSQR: An algorithm for sparse linear equations and sparse least squares, ACM transactions on Mathematical Software, vol 8, no 1, p 43-71

Paige, C.C. and Michael A. Saunders, 1982 Algorithm 583, LSQR: Sparse linear equations and least squares problems, ACM transactions on Mathematical Software, vol 8, no 2, p 195-209

Thorson, J.R., 1984, Velocity and slant stack inversion: PhD thesis, Stanford University. Also published as SEP-41 and submitted to Geophysics, in press.